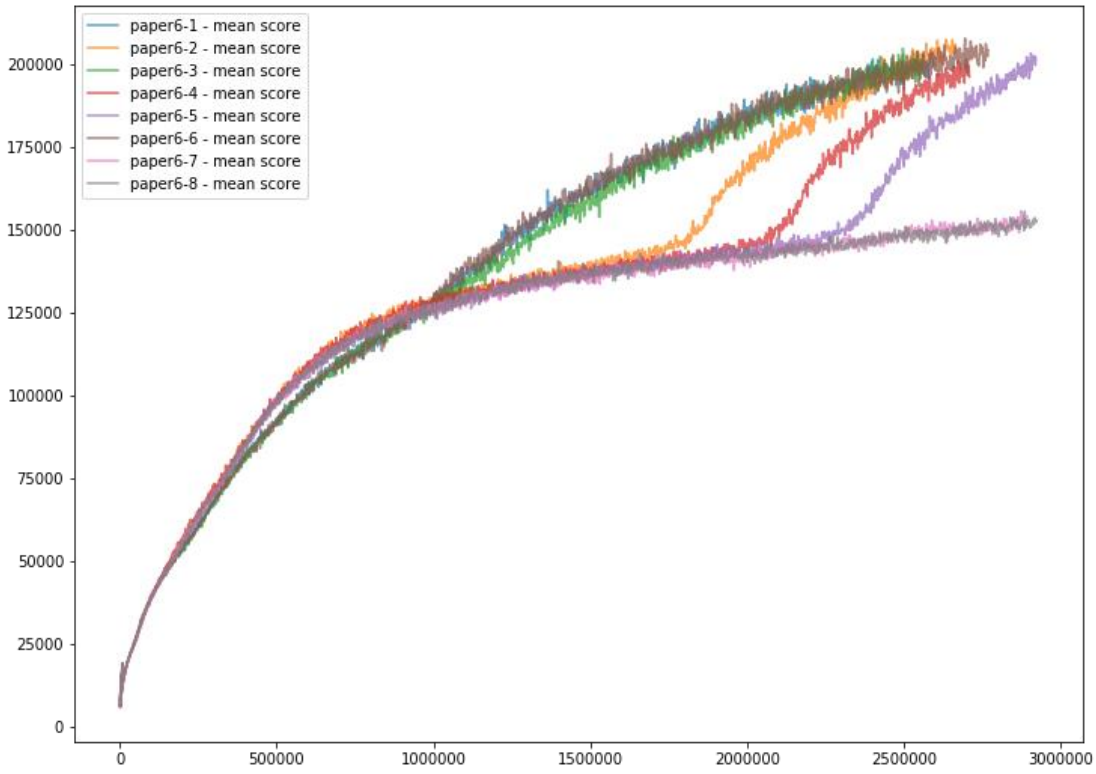
DeepLearning Lab5 Report

Chien-Hsun Lai (0656078)

Github: https://github.com/jxcodetw/TD-Learning-2048

Results (8 different random seed):

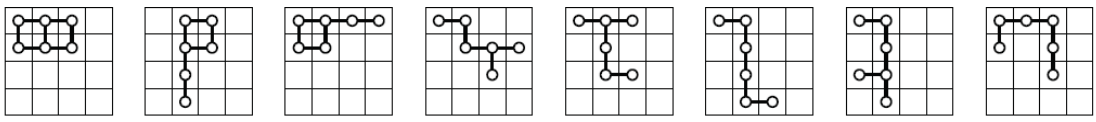



```
Step: 2768000
Mean Score: 204268.56, Max Score: 384448
  256:  100.0%  (0.1%)
  512:  99.9%   (0.4%)
 1024:  99.6%   (1.9%)
 2048:  97.7%   (6.8%)
 4096:  90.9%   (10.5%)
 8192:  80.4%   (34.7%)
16384:  45.8%   (45.8%)
```

Algorithm: TD-afterstate

Learning rate: 0.02;

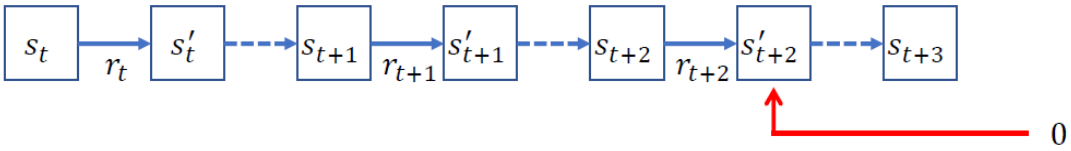
Tuple: use the shape from [Oka et al].



Explain the mechanism of TD(0)

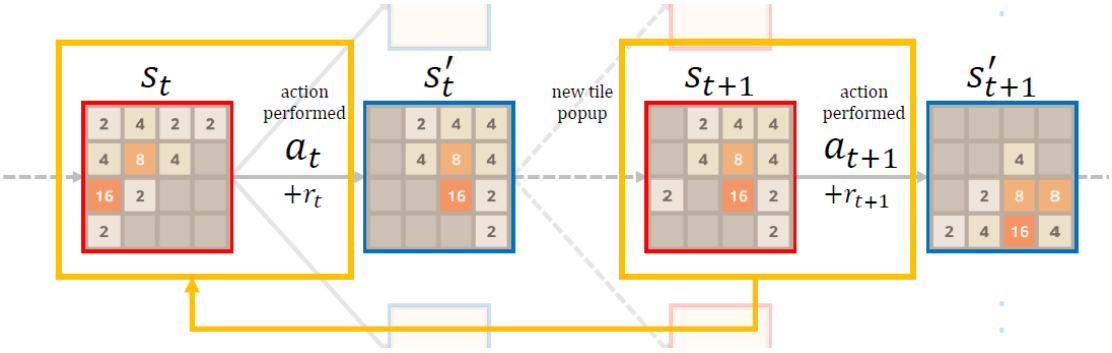
TD(0) uses a value table to estimate the expected reward after a given state. In this lab, we use n-tuple network to replace the table because the whole state of the game is too big. N-tuple network can efficiently approximate the true value table, because of the shape it considered. TD(0) first plays an episode of the game and record the trajectory. Then it learns from that trajectory by updating it's value table. The estimate near the end of the game is the most accurate estimate. After a few learning, the true value of the state can be more accurately estimate due to the state after it is getting more accurate.

For example:



If S'_{t+2} is the terminal state, then we can know that the value of S'_{t+1} is r_{t+2}+S'_{t+2} And because we definitely know S'_{t+2} is zero, so we know the value of S'_{t+1}. And as we are getting to know the value of the states near the end, we can estimate the states before them more accurately. Finally propagate to the state near beginning. We can know to value of each state.

Describe how to train and use a V(state) network

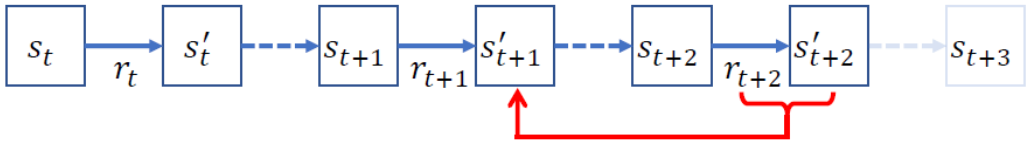


Because we only has V(before-state) so we have to evaluate all possible V(before-state), and take the expectation of it (mean score) so we can know which action will lead to the best condition.

When updating, we update St with St+1 + Rt+1

Describe how to train and use a V(after-state)

network



S_{t+2} is randomly popup (2 or 4 and different position), so V(S'_{t+1}) will automatically become more accurate. If the ai has played many times to that state. It works like the spirit of Monte-Carlo. The red line indicate the target value, we will update V(S'_{t+1}) to those states it become.

When using this network, we can simply chose whatever action that has max V(after-state).

Describe how the code work

The entry point of my program is main.cpp. It will new a “Agent” class that contains a game inside and the tuple network. In the main loop, it calls Agent::play_game() to simulate an episode of a game and return the states it went through. Game class will always maintain the four possible move’s after state. In each turn the agent will check each move. If the before state is equal to the after state, it won’t consider that move. If all move are invalid (before == after), it will terminate the game. If there are still moves it can take. It will choose the best estimate value one using current n-tuple network. Then it will update the game to that move and record that move into history. After the game is finished. The history is return.

In Agent::learn_backward(), it update from the last state.

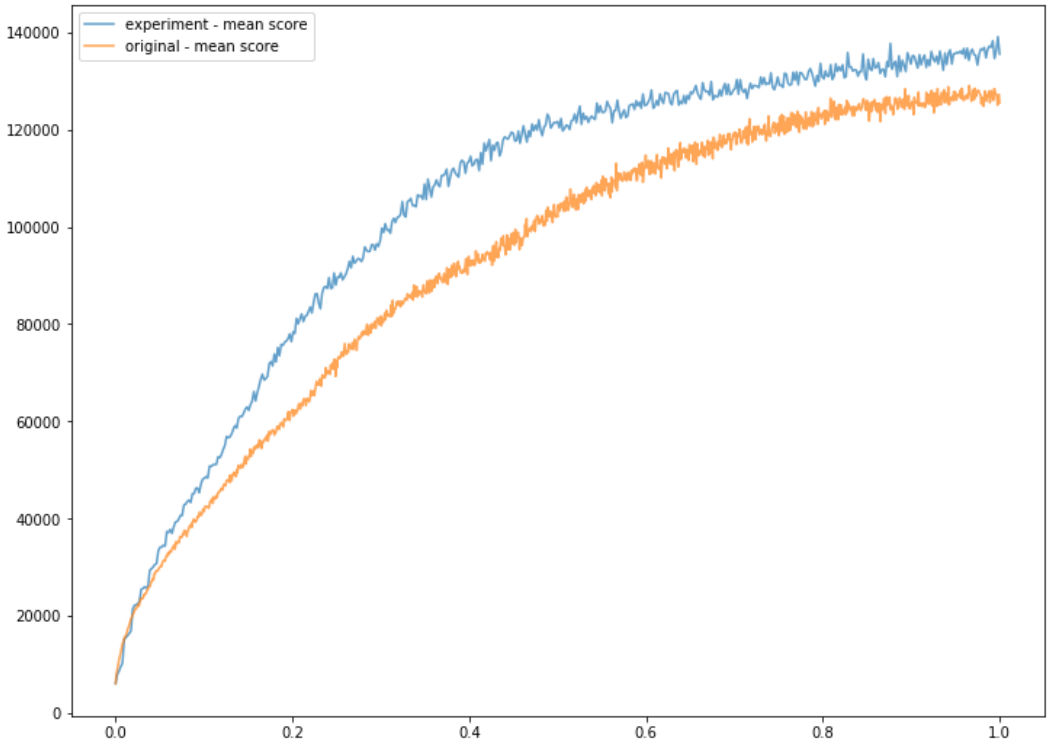
```cpp
std::vector<Move> Agent::play_game() {
        static std::string mapping[] = {"up", "right", "down", "left"};
        std::vector<Move> history;
        game.init();
        while (!game.is_terminate_state()) {
                Move *best_move = &(game.move[0]);
                for(size_t i=0;i<4;++i) {
                        if (game.move[i].is_valid()) {
                                game.move[i].value = (float)game.move[i].reward + tuple_net.V(game.move[i].afterState);
                                if (game.move[i].value > best_move->value) {
                                        best_move = &(game.move[i]);
                                }
                        }
                }
                Move best = *best_move;
                game.take(*best_move);
                history.push_back(best);
        }
        return history;
}

void Agent::learn_backward(std::vector<Move> &history) {
        for(size_t i=0;i<history.size();++i) {
                Move &h = history[i];
                h.value = (float)h.reward + tuple_net.V(h.afterState);
        }
        float last = 0;
        while(!history.empty()) {
                Move &h = history.back();
                last = h.reward + this->tuple_net.update_V(h.afterState, this->alpha * (last - (h.value - h.reward)));
                history.pop_back();
        }
}
```

Other improvement (main_experiment.cpp)

Notice that learn_backward_review() first re-evaluate the value of each move. This is because the modification I made to the learning algorithm.

The original algorithm play one episode and learn one episode. Mine does that too, but every 10000 episode it will learn from the history in the past 10000 episode. It works like reviewing the path it take. Because simulating a game is slow. So I store the histories. So that it can random sample the history to update it's network more and faster.

```cpp
if (i % review_interval == 0) {
    std::cout << "learning" << std::endl;
    for(int bd=0;bd<10;++bd) {
        std::cout << "random sample learning" << std::endl;
        for(int k=0;k<review_interval;++k) {
            std::vector<Move> hhh = pool[rand() % review_interval];
            ai.learn_backward_review(hhh);
        }
    }
    pool.clear();
    std::cout << "learned" << std::endl;
}
```



The x axis is the time it takes (proportional), so in the same time the experiment version learns faster. But it seems that they will remain small difference in the end.

```
Mean Score: 138506.28, Max Score: 315908
  512:  100.0%  (0.4%)
 1024:  99.7%   (1.3%)
 2048:  98.3%   (6.9%)
 4096:  91.4%   (14.4%)
 8192:  77.1%   (74.5%)
16384:  2.6%    (2.6%)
```

(experiment method)

```
Mean Score: 128095.38, Max Score: 310544
  512:  100.0%  (0.3%)
 1024:  99.7%   (1.9%)
 2048:  97.8%   (9.6%)
 4096:  88.2%   (19.0%)
 8192:  69.2%   (67.9%)
16384:  1.4%    (1.4%)
```

(original method)

Performance

My implementation is faster than example code under same hyper parameter and compile flags same amount of episode.

TA's: 181.25s user 0.06s system 99% cpu 3:01.36 total

My: 22.85s user 0.13s system 99% cpu 22.993 total